NOTES ON
THE 4100
FORTRAN
COMPILER

NØTES ØN THE 4100 FORTRAN CØMPILER

Contents

1. The Fortran Language

History

The first version of Fortran was implemented on the IBM 704 in 1956. This version, known as Fortran I, contained no procedure facilities, was optimising and very slow. It was soon expanded to Fortran II which lasted right up to 1964. Fortran II did contain procedure facilities but was still nowhere as sophisticated as Algol.

In 1964 Fortran IV was introduced and most of the programmer's objections to Fortran were removed. The extra facilities provided included complex, double precision and logical variables. Since the introduction of Fortran IV there has been an explosion in its use and many people are of the opinion that all scientific computers should be supplied with a Fortran compiler and that the Algol compiler should be an optional extra.

Program Structure

No attempt is made here to give the precise details of the Fortran language, only the general structure and a brief mention of the different statement types.

Fortran consists of statements arranged in subprograms. Each statement is written as a newline of up to 80 characters or a card of 80 characters. The first 5 characters are reserved for the statement label (if any). The statement label must be a number. Characters 73-80 are ignored by the Fortran compiler but may be used by the programmer for card numbering. The Fortran statement itself is written in free format in character positions 7-72. If a statement is too long it may be continued on any number of continuation lines. A continuation line has a non-zero character in character position 6.

There are 3 types of subprograms:

a) Main program.

The first executable statement of the main program is the first one obeyed. It contains the STOP statement which terminates the program.

b) Subroutines.

Subroutines have the following form:-

```
SUBROUTINE SUBR (    )
RETURN
END
```

A subroutine is entered from another subprogram by the statement:-

```
CALL SUBR (   ).
```

The RETURN statement causes control to be transferred to the statement following the CALL statement. A subroutine does not have to have any arguments. Any input to or any output from a subroutine is done using arguments.

c) Function Subprogram.

Function subprograms have the following form:-

```
FUNCTION   FUN (   )
FUN =
RETURN
END
```

A function subprogram has a value associated with it, and must have at least one parameter as an argument. The function subprogram is brought into operation by writing its name with the required arguments whenever its value is required in another subprogram.

The last statement of a subprogram is an END statement which tells the compiler to stop the compilation of that subprogram. Subprograms can be compiled independently and the object code stored on magnetic tape. Communication between subprograms is limited to either by parameters or by the COMMON statement.

Subprogram Structure

Each subprogram begins with a specification statement area. This area contains non-executable statements which supply the compiler with information concerning variables. The following are various statement types appearing in the specification statement area.

```
REAL A
INTEGER B
COMPLEX C
DOUBLE PRECISION D
LOGICAL L
DIMENSION I(20)
COMMON X,Y,Z
EQUIVALENCE (Z, I(7))
EXTERNAL FUN
```

There is no need to declare every variable since there is implicit typing. If a variable does not appear in a type statement then if the first letter of the variable name is I, J, K, L, M or N then the variable is of type integer otherwise it is of type real.

After the specification statement area comes the statement functions. Statement functions are called in the same way as function subprograms. The difference is that the definition of the function consists of a single statement in the subprogram in which the function isneeded.

After the statement functions comes the program body containing the executable statements which are listed below.

1) Arithmetic Statement. These can contain function calls.

2) Arithmetic IF statement

   IF (arithmetic expression) L1, L2, L3.

   If the arithmetic expression is negative, zero or positive then the next statement obeyed is the one with label L1, L2 or L3 respectively.

3) Logical IF statement

   IF (logical expression) statement

   The statement must not be another logical IF statement. If the logical expression is true the statement is obeyed and if the logical expression is false then the statement is not obeyed.

4) GØTØ L

   The next statement obeyed is the one with label L.

5) Computed GØTØ statement

   GØTØ (A,B, ,C), I

   The next statement obeyed is the statement with the label which is the Ith label in the list of labels enclosed within brackets in the GØTØ statement.

6) Assigned GOTO statement.

   ASSIGN 23 TO I

   GOTO I, (21, 22, 23)

   The next statement obeyed after the GOTO statement is the statement which has the label which has the value of I.. This value must appear in the list of labels enclosed within brackets in the GOTO statement.

7) The DØ statement.

DØ 10 I = L, M, N

10 . . . . . .

The statements from the statement after the DØ statement up to and including the statement with label 10 are repeatedly executed for I = L, I = L + N, I = L + 2N and so on until I has its largest value that does not exceed M.

8) Input & Output statements.

The input and output statements are of the form

READ (device no. , FORMAT statement label) list

e.g. WRITE (2,17) X,Y, (A(I), I=1,100)

The FORMAT statement is a non-executable statement giving the description of the variables and character strings to be input or output. It is of the following form 17 FORMAT (list of descriptions)

e.g. 17 FORMAT (2F10.3, 10 (/10F8.2))

9) The CALL statement will bring a subroutine into use.

2. 4100 Fortran Design Objectives

When the design objectives for 4100 Fortran were put forward Elliotts had an agreement with NCR in which NCR dealt with the commercial aspects, and Elliotts with the scientific aspects, of computing. Consequently 4100 Fortran was designed for scientific users, in particular, universities and research associations. This type of user has a large number of programs to be run in a short time but also has a few large programs.

The original design objectives are listed below:-

Fast compilation.
"Load and Go".
Batch Processing.
Library of Binary Programs on Magnetic Tape.
Independent Compilation of Subprograms.
Compact Object Code.
Small "System" (e.g. Device Routines).
Segmentation.
Neat Subprograms.
Full A.S.A. Language.

The first 4 are the main requirements for a university user and the next 4 are the main requirements for running very large programs.

Optimisation projected is not yet implemented for 4100 Fortran although IBM on their larger machines use optimisation techniques. However efficiency can be increased in 4100 Fortran by using Neat subprograms, which can also be used to provide capabilities beyond the Fortran language itself.

The GAMM measure of efficiency, which gives the time in microseconds to do a useful floating point operation, can be used to compare the efficiencies of different compilers for different languages. Below is a table of such comparisons for the 4100, the figures being given as percentages. Machine code is taken to be 100% efficient.

	4130	4120
Machine code	100	100
Fortran	78	63
Algol	55	45

These figures reflect the fact that Fortran is an easier language to compile than Algol. Fortran was in fact designed so that it would be easy to compile. Algol also has automatic subscript checking.

Below is a brief summary of the main achievements so far. The object code has been kept compact and compares very well with other Fortran compilers. On average the object code produced is 12 words per Fortran statement. The following table gives figures comparing the object code produced by other compilers taking the 4100 Fortran value as unity.

SDS 9300	4100	All others
0.3	1	1.1 to 1.3

The system for B20 is 5,690 words of store and for T300 is 10,000 words of store. The system at the moment is not yet modular. Almost all the ASA Fortran language has been implemented. In 4100 Fortran it is not possible to mix up EQUIVALENCE and FORMAT statements. Complex and double precision working and the DATA statement have yet to be incorporated into the compiler.

Load and GO is not very good on 16k words of store. Batch processing has been achieved by having control statements in the source program. A Neat subprogram must first be compiled to paper tape using NEATOUT and when needed the I/C paper tape is input and the subprogram is stored on magnetic tape. Segmentation is under the control of the programmer by the use of the SEGMENT statement in the source program between subprograms.

3. Multipass Structure

The Elliott 503 Fortran compiler takes up 12k words of store, with a word length of 39 bits. A rough estimate of the length of the 4100 Fortran compiler can be made if a bit in each computer has the same value. The estimate so obtained is 20k words of store. Since we wanted the 4100 Fortran compiler to work with 16k words of store it was decided to split the compiler up into 2 passes which would not be held in store at the same time. As it happened the 4100 Fortran compiler is 9k words of store, and with both the system and compiler in store this still leaves some free store on a 16k 4100. So the splitting up of the compiler into 2 passes was not strictly necessary but has proved very convenient.

Pass 1 takes up 5k words of store. Its main functions are to input the source code, to perform the syntactical analysis and to produce the intermediate code. It also performs the first half of any operation which is naturally 2 pass, such as arithmetic expressions which are converted by Pass 1 to reverse Polish form. Pass 2 takes the intermediate code and from it produces object code. Pass 1 takes 40% of compile time while Pass 2 takes only 8% of the compile time, leaving 52% for the input of the characters.

The order in which these passes are made is as follows. On the B20 system and also the T30C system with 24k words of store or more each subprogram is processed by both passes of the compiler before the next subprogram is processed. On a T30C system with less than 24k words of store each subprogram in a program is processed by Pass 1 of the compiler before any of the intermediate code is processed by Pass 2.

Intermediate code is read in backwards by Pass 2. The original reason for this was that it was thought that the output from the intermediate code would have to be output to paper tape. Thus since the dictionaries are output after the intermediate code by Pass 1 but are needed before the intermediate code by Pass 2, the paper tape would have had to been read in by Pass 2 backwards. The situation now is that Pass 1 builds up intermediate code from the bottom of free store and if it is a large store Pass 2 reads the intermediate code from the store while if it is a small store the intermediate code is written backwards to magnetic tape ready to be read by Pass 2 at a later date. Pass 2 builds down object code from the top of free store.

An advantage of reading the intermediate code backwards is in the compilation of conditional jumps (always forwards on 4100) since Pass 2 will meet the label before the jump instruction. This means that the destination of the jump is known when the object code for the jump is issued. Another advantage is for the projected optimisation of the object code produced by DO loops. Certain calculations (generally of an array access nature) are removed from the inside of the DO loop and placed in front of the DO loop. To do this the output of the relevant parts of the object code is just delayed.

4. Pass 1

4.1 General Structure

The general structure of Pass 1 is that the Neat blocks are arranged on 4 levels. At the first level is BLOCK NEXT which recognises the different statement types.

A symbol is the next logical constituent of a statement above that of the character. This is illustrated in the following 2 statements in which each symbol is underlined separately.

DO I = 1 , TOPLIM
X = A (1) + 0.3 * (-2.0E-10)

To determine the type of a statement it is not sufficient just to look at the first symbol of a statement as the 4 statements will show.

```
DO 3 I = 1,2
DO 3I = 12
FORMAT (5H) = A(1)
FORMAT (I) = A(1)
```

The 2nd and 4th statements are both assignment statements while the first is the beginning of a DO loop and the 3rd statement is a FORMAT statement for transferrin the 5 characters ") = A (1 ".

If a statement has an equals sign on level zero i.e. not enclosed in brackets then the statement is an arithmetic or a DO statement. Block NEXT checks FIELD to decide whether the statement appears in the specification statement area, function statement area or the main body of the subprogram.

The second level of Neat blocks contains a separate block for each statement type and when BLOCK NEXT has determined the type of a statement a jump is made to the appropriate block. These blocks in the second level produce the intermediate code and perform the syntactical analysis.

The third level of Neat blocks contains 4 utility blocks which are entered from blocks in the first and second levels. The first block provides the dictionary routines, the second the intermediate code handling routines, the third is for symbol recognition and the fourth is the error block.

The fourth level of Neat blocks contains a block to input characters not from the input device but from a buffer holding a whole line and also to scan ahead to see if the next line is the same statement. This block is mainly used by the block which recognises symbols.

The block which recognises symbols operates on 3 levels. The first level just hands over characters one at a time ignoring spaces, except within Hollerith strings. The second level hands over symbols. If the symbol begins with either +, -, * or / then the symbol is an arithmetic operator. If the first character of a symbol is alphabetic then the symbol is either an identifier, a function name or a word giving the statement type. If the first character of a symbol is numeric then the symbol is a constant. The following 2 character strings illustrate possible trouble in identifying the symbol.

345. E-10

345. EQ.

When the full stop is met this could either mean that it was a decimal point in a constant or the beginning of a logical operator. To decide which case it is the block scans ahead. If in the 2nd case the last 3 characters are on a continuation line then by the time it has decided which of the 2 cases it is the continuation line has overwritten the previous line in the buffer and the point where the processing had got to has been lost. Thus a restriction must be imposed that certain symbols must not be on more than one line. When this level hands over a symbol it also hands over a number giving the type of the symbol.

The third level is given a symbol and comes back with whether it is an identifier or an array. In the specification statement area the general structure is that of a word followed by a list of symbols separated by terminators. If an error has occurred in, say, an array declarator then an error message will be output every time that array is referenced. So if the error occurred in the declaration with respect to only one array then we want the compiler to read until the next terminator and then to continue as though the error had not occurred

Since it is very much easier and quicker to work with integers, integer constants are manipulated as integers. If an integer overflows it is converted to a floating point number without any loss of accuracy. Similarly if a floating point number overflows it is converted to a double precision number.

The compiler has its own workspace in which there is a dynamically used stack for holding information about COMMON blocks and DO loops. This stack is 256 words long and is used from both ends. The free space in the middle is used by Pass 1 in the compilation of arithmetic expressions.

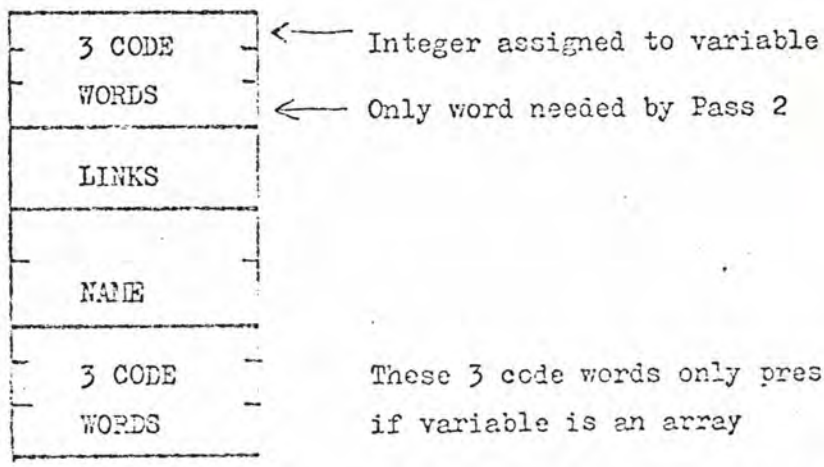
Pass 1 builds up intermediate code from the bottom of the free store area and the dictionaries are built down from the top in 256 word blocks. The area in between the intermediate code and the dictionaries is used to hold equivalance information.

4.2 Intermediate Code

The main purpose of Pass 1 is to output intermediate code (I/C) which should be as short and as compact as possible. The intermediate code is read by Pass 2 backwards.

The I/C for a subprogram begins with the subprogram head giving the name and class of the subprogram. After this comes the dictionaries followed by about 10 words of information needed by Pass 2, such as the number of variables and the number of DO loops in the subprogram. This is then followed by the main body of the I/C which is a continuous binary stream.

There are dictionaries for labels, variables, constants, COMMON block names and subprograms called. A Pass 1 dictionary entry for a variable has the following format:



A variable is given a name by the programmer. Pass 1 assigns an integer to each variable in such a way that the nth variable in a subprogram is assigned the integer n. When Pass 1 wants to access the information concerning variable number n the nth half word in a table contains a pointer to the information.

This method is a straight look-up and does not necessitate any dictionary searching. At the Pass 2 stage the entries in the dictionaries are now in numerical order, so that when Pass 2 wants to access the information concerning variable number n it just jumps to the nth variable in the dictionary.

In the I/C we want the ability to represent integers up to 1023. If the number is represented as a binary number then every number from 0 to 1023 occupies 10 bits. Since most of the integers we want to represent are relatively small this method is not very compact. The method we adopted is as follows. If the first bit is 0 then the integer is between 0 and 31 and the next 5 bits are the binary representation of the integer. If the first bit is 1 then the integer is between 32 and 1023. In this case if the second bit is 0 then the integer is between 32-63 and the next 5 bits are the binary representation of the integer minus 32. If the second bit is 1 then the next 10 bits are the binary representation of the integer. The average number of bits needed to represent an integer is now about 7 with a subsequent reduction in the length of the I/C.

Below is the general structure of the I/C issued for a statement as seen by Pass 2. If the first bit is 0 then the statement has no label and if it is 1 then the statement has a label and the 1 is followed by the coded label integer. The bits which follow this give information concerning DO loops. For each DO loop ending on this statement there is a sequence of bits giving certain information about the DO loop. Each sequence is preceded by the bit 1 and the last sequence is followed by the bit 0.

The next section of bits gives the statement type code. If the first bit is 0 then the statement is arithmetic. If the first bit is 1 then the next 5 bits give the statement type. We used this method of coding the statement type because the arithmetic statement is the most common. On average we need $3\frac{1}{2}$ bits to give the statement code while a fixed length code would need 5 bits and an optimising code uses about $2\frac{1}{2}$ bits.

After the statement type code is the I/C for the statement itself followed by the code for an end of a statement.

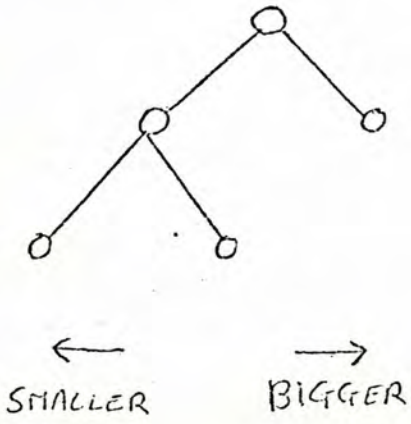
When Pass 1 meets a variable name it wants to know if it has met the variable before. If it has met the variable before then it will want to be able to access all the information that has been stored away concerning the particular variable. If it has not met the variable before then it will want to store all that is known about the variable.

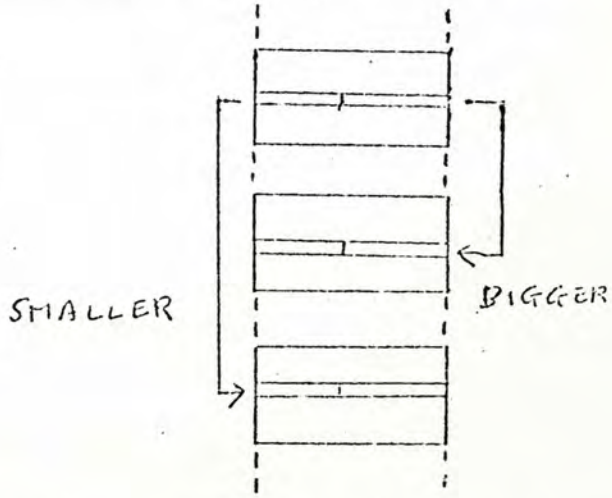
The properties of a good dictionary are that it is compact and quick to use. In a linear dictionary all the entries are listed one after the other. When it is asked to find a particular entry it searches through the dictionary from the beginning until the entry is found. If the entry is not found then another entry is inserted at the end of the dictionary. A linear dictionary is extremely slow since on average about half the entries in the dictionary have to be examined every time any information is accessed.

A very much faster dictionary can be constructed using scatter functions. The entries are again listed one after the other but they are accessed in a completely different manner. At the beginning of the dictionary is a table of length 128, say. When the dictionary routine is given a variable name it will use the name to construct a number n, say, between 1 and 128. The nth position in the table will hold a printer to the first entry of a linear list of entries. Although the entries in this list have to be examined in turn when searching for a particular entry, the scatter function dictionary is very much faster than a linear dictionary since the 128-hold branch has reduced the number of examinations by a factor comparable with the size of the table. It is, of course, possible to nest scatter functions by having the pointer from one table pointing to another table containing more pointers.

We, however, in our Pass 1 dictionaries used tree structures. We now wish that we had used scatter functions but we thought that we would have to pass over the dictionaries to Pass 2 in a different form than is used now. Nevertheless our tree structures dictionaries are quite fast, and our compiler spends about 6% of its time in its dictionaries. If we had used scatter function dictionaries the time spent in the dictionaries would have been about 2%

Below are 2 diagrams illustrating the structure of our tree dictionaries, the one on the left the logical structure and the one on the right the physical structure.





The dictionary entries are arranged in a linear list but in searching for a particular variable name the entries are not examined one after the other down the list. When the first entry has been examined a branch takes place according to whether the variable name being searched for is bigger or smaller than the variable name of the entry being examined. Every entry in the dictionary has a similar branch. The branching is achieved by 2 $\frac{1}{2}$-word pointers held in each dictionary entry. As dictionary space is allocated in 256 word blocks the least significant 4 bits give the block in which the destination of branch lies while the remaining 8 bits give the offset within that block. If a particular pointer is zero then that branch does not exist, which means that the variable has not been encountered before and the dictionary routine will then insert a dictionary entry for this variable, thus creating a new branch. The pointers in this new entry will both be initially set to zero, while the pointers in the previous entry, which were zero will be updated to point to the new entry.

A variable name is stored in 2 locations with 4 characters to a location, each character represented by its internal code value. By saying that 1 variable name is less than another variable name we mean that the 1st location of the 1st variable name holds a smaller integer than the 1st location of the 2nd variable name, or if these 2 integers are the same then the 2nd location of the 1st variable name holds a smaller integer than the 2nd location of the 2nd variable name.

The average number of examinations needed to find a particular dictionary entry is (R $\log_e$ N + S) where N is the number of entries in the dictionary, and R and S are constants.

So far we have discussed 2 functions of the dictionaries. The first is to search the dictionary for a particular entry and exit with the required information. The second is to insert a dictionary entry if the first function fails to find the entry it is looking for. There is another function which takes place when the dictionaries are complete, and involves searching through the dictionary and exiting with information such as which labels have not been defined or how many variables have certain bits in a particular codeword set.

Below is the code word structure for the variable, array and function dictionary entries.

Codeword 1

Bits 1-2	Used by dictionary routine.
Bits 3-5	Gives type i.e. Real, Complex etc..
Bits 6-11	If array gives number of subscripts. If function or subroutine gives number of arguments.
Bit 12	Indicates whether parameters are virtual or not.
Bits 13-14	Gives type of workspace i.e. whether normal or abnormal workspace.
Bit 15	Historical.
Bits 16-21	Free.
Bits 22-24	Gives class of the entry i.e. whether identifier, array, function, basic external function etc..

Codeword 2

Bits 1-15	Used if item is in abnormal workspace. If it is in COMMON W/S then gives offset to start of COMMON block. If it is in array W/S gives offset to start of entire array W/S.
Bits 16-24	If in COMMON W/S gives the COMMON block number.

There are 2 types of abnormal W/S. The first is COMMON W/S which is used by an item which is in a COMMON block or which is equivalenced to something in COMMON.

Codeword 3

Bits 1-15 Gives the integer reference of the item which will be passed over to Pass 2. As soon as the variable dictionary is complete the variable name is no longer needed.

Codewords 4, 5, and 6

Bits 1-15 Gives the maximum size of the 3 possible dimensions of an array. These 3 codewords are only present in array dictionary entries.

Specification statements do not cause any intermediate code to be issued but provide information to update the dictionaries.

4.4 COMMON and EQUIVALENCE statements.

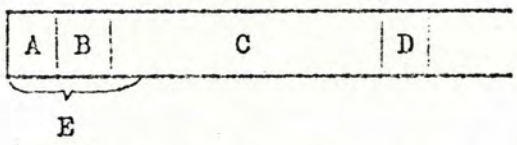
In Fortran II COMMON statements were relatively simple and of the following form

```
COMMON  A, B, C(100), D(3,4)
```

The effect of this statement was to assign workspace for these identifiers which would be common to all the subprograms. If another subprogram contained the following statement

```
COMMON  E(10)
```

then the beginning of the array E would coincide with the variable A as illustrated in the following diagram.



Fortran IV has 2 types of COMMON statement. The first is the Fortran II COMMON statement while the second has one of the following forms.

```
COMMON / BLOCK 1 / A, B, C(3) /
COMMON / BLOCK 1 / D, E, F(3) /
COMMON / / G, H, I(3) /
```

The effect is very similar to the first type of COMMON statement except that the W/S common to all the subprograms is now allocated in blocks. In the case of the first 2 statements they are known as named COMMON blocks and in the case of the last statement, unnamed COMMON blocks. If the first 2 statements appear in 2 different subprograms then the 2 blocks must be of the same length, and the identifiers A and D, B and E, and C and F will occupy the same locations respectively. Unnamed COMMON blocks do not have to be of the same length.

The EQUIVALENCE statement is used to enable the programmer to make several variables share the same W/S, and has the following form.

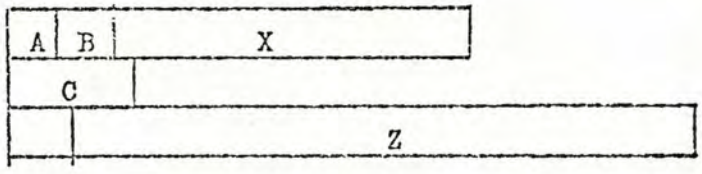
```
EQUIVALENCE (A, B, C(1)), (D, C(2)), (E, C(3))
```

The effect of this is to make A, B and C(1) share the same W/S, D and C(2) the same W/S, and E and C(3) the same W/S.

The COMMON and EQUIVALENCE statements have a significant interaction which is illustrated by the following 3 statements.

```
COMMON        A, B, X(10)
DIMENSION     C(3), Z(20)
EQUIVALENCE   (B, C(2)), (X(1), Z(1))
```

The effect of the EQUIVALENCE statement is to lengthen the COMMON block as shown in the following diagram.



Hence it is not possible to know the length of the COMMON blocks until all the EQUIVALENCE statements have been processed.

The EQUIVALENCE and COMMON statements do not produce any code, and the information from the statements is stored in 2 stacks as each statement is met. The COMMON stack has a section for each COMMON block and bit 24 is set in the word containing the block name. The EQUIVALENCE stack has a section for each sublist and the last word in each section has bit 24 set.

At the end of the specification statement area the information in the 2 stacks is the processed in the following 6 stages:

1. For each sublist in the EQUIVALENCE stack, the first item is taken as a base and the offset of the other items in that sublist are worked out relative to that base.

2. In each COMMON block the offset of each item is worked out relative to the beginning of that block. The lengths of the COMMON blocks are still not known at this stage.

3. The items in each EQUIVALENCE sublist are reordered so that the base is now the lowest addressed item. This means that all the offsets are now positive with respect to the new base.

4. The EQUIVALENCE sublists are now combined so that every item is in a unique sublist. It is now possible to spot inconsistencies such as

   EQUIVALENCE (C(3), D(4)), (C(1), D(5))

5. Now chunks of W/S are assigned for each EQUIVALENCE sublist. If none of the items in the sublist are in COMMON then W/S is allocated in the ARRAY area. If one of the items in the sublist is in COMMON then the whole sublist is put in the relevant COMMON block. A check is now made to see if there is more than 1 item in COMMON, which is not allowed. Another check is made to make sure that a COMMON block is not extended backwards.

6. Now that all the COMMON block lengths are known, the blocks are strung together and put in COMMON W/S.

4.5 Arithmetic and Logical Expressions.

In the past Elliott compilers have compiled arithmetic expressions by a direct recursive procedure. This would try to evaluate the expressions on either side of an operator. If however these expressions contained a further operator then it would try to evaluate the expressions on either side of this operator, and so on.

In our 2 pass Fortran compiler we use the first pass to transform the source code into reverse Polish form and the second pass to produce object code from the reverse Polish. This second stage is fairly simple and tends to produce more efficient object code than the recursive method. The first stage also includes the syntactical analysis. Before the reverse Polish is output to the intermediate code its order is reversed so that when Pass 2 reads the intermediate code backwards it does in fact read the reverse Polish forwards.

Below are 2 examples of arithmetic expressions followed by their reverse Polish forms:

A * (B + C)	ABC + *
A * B + C	AB * C +

The reverse Polish contains no brackets and to convert back to the original arithmetic expression the following rule is applied repeatedly - the first operator operates on the previous 2 items.

The algorithms for converting to reverse Polish are quite simple and depend on the following order of precedence:

1. =
2. . OR .
3. . AND .
4. . NOT .
5. . EQ. , .NE. , etc.
6. + , binary minus.
7. * , /
8. unary minus.
9. **

If 2 operators have the same precedence then precedence is taken from left to right. This means that A/B * C would be evaluated as (A/B) * C. Functions, arrays and the equals sign are treated as operators.

The conversion algorithm uses a pushdown stack and the various constituents of an expression are dealt with as follows.

(i) Simple operands such as variables, constants, function names used as parameters, but not arrays or functions with arguments, are added to the output string.

(ii) When an operator is met, the stack is popped up until an operator of lower precedence, a function, array or left bracket is met. This is left in the stack, the items popped up output to the output string and the new operator pushed down onto the stack.

(iii) Functions with parameters and arrays with subscripts are pushed down in the stack and the immediately following left bracket ignored.

(iv) Left brackets not immediately after a function or array name are pushed down in the stack.

(v) When a comma is met the stack is popped up until a function or array name is met, this being left in the stack.

(vi) When a right bracket is met the stack is popped up until a function or array is met, this also being popped up, or until a left bracket is met, which is then thrown away.

(vii) When the end of the expression is met the stack is popped up until it is empty.

During this conversion the DSTACK is used. The push down stack is built down from its top while the output string is built up from its bottom. When this string is output to the intermediate code it is the last item to be added to the string which is handed over first. Flags are used to check such things as whether an operand follows an operator, whether there is a 2nd equals sign or whether there is an operator on the L.H.S of an equals sign.

During the conversion to reverse Polish a check is made of the number of subscripts in arrays using a small stack known as ESTACK, Since it is possible to nest arrays, as in A(B (C (1,1) , 1) , 1,1), the number of subscripts must be checked on each level. When an array name with n subscripts is met then n-1 is pushed down in ESTACK. When a comma is met the active top is decremented and when a right bracket is met the top of the stack is popped up and checked against zero. If a function has unknown number of parameters then 64N + 63 is pushed down onto the stack. When the right bracket is met the top of the stack is far too big and from the number of times that it has been decremented it is possible to determine the number of parameters of the function. (Here N is a pointer at the dictionary to allow the entry there to have the number of parameters defined when it becomes known).

When the reverse Polish string has been formed it is scanned from the beginning to check the types of the variables in relation to the operators since the following assignment is not allowed:

J = I + B

where B is logical and I, J are integers. Another push down stack is used for this purpose. If the types of the variables on either side of an arithmetic operator are different then the result has the type of the higher of the 2 in the following list.

COMPLEX
DOUBLE PRECISION
REAL
INTEGER

If the types of the arithmetic variables on either side of an equals sign are different we allow any meaningful unambiguous assignment. Below are 2 examples:

COMPLEX = REAL
REAL = COMPLEX

The 1st assignment is allowed and the complex variable has the real variable for it real part and zero for its imaginary part. The 2nd assignment is not allowed since the real variable could either be the real part or the absolute magnitude of the complex variable.

4.6 FORMAT Statements

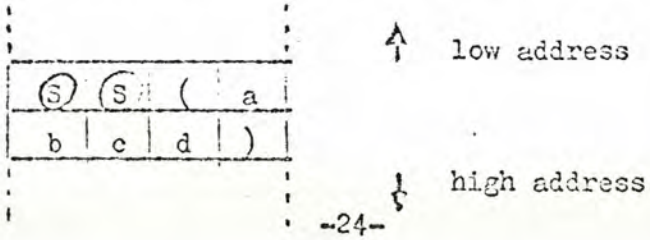
The FORMAT statement is a non-executable statement referenced by a READ/WRITE statement to control the input and output of information.

The action of Pass 1 is simply to hand over one by one the characters between and including the outside pair of brackets to the intermediate code as 6 bit internal code characters.

The action of Pass 2 is to pick up the characters from the intermediate code and pack them in the constants block. If the FORMAT statement was

FORMAT (a b c d)

then the 6 characters would be packed 4 to a word as shown below:



The object code issued for a FORMAT statement would be of the form.

```
JF      4
LDR:L   <address of FORMAT list>
JI      0
```

If the FORMAT information is to be input at run time into an array then the 2nd instruction would be

```
LDR:L   <address of location holding array ZERO>
```

while if the READ/WRITE statement was unformatted (i.e. transfer to be in binary) then the 2nd instruction would be

```
LDR:L   0
```

The object code issued for a READ/WRITE statement would be of the form

```
LD:L    <address of parameter list>
JFL     <FORMAT statement code>
JIL     <READ/WRITE Dynamic Routines>
```

The 2nd instruction jumps to the 2nd instruction of the FORMAT statement code and jumps back with the relevant information in the R register. The JIL instruction jumps to the Dynamic Routines which perform the information transfer. When the FORMAT statement code is met in the execution of the program at run-time, the FORMAT code is not obeyed but is jumped round by the JF instruction since the FORMAT statement is non-executable.

5. Pass 2

5.1 General Structure

Pass 2 inputs the I/C from Pass 1 backwards, processes it and then outputs the object code in a stream of blocks of code, constants, relocation markers, switch tables and subprogram heads.

When Pass 2 issues the object it does not know where any of the workspace will be in store but knows the position of a variable relative to the beginning of the workspace for that block. Thus the code

```
LD      X
LD:L    2
```

would be compiled as

```
43200007
43000002
```

which is known as relocatable object code. Just before the program is run the position in store of the workspace is known and the address of the beginning of the workspace is added into the instruction so the 2 orders above would look like

```
43210007
43000002
```

if the block of workspace containing the variable X starts at address 8096.

Not only variables but also constants, dynamic routine entries and COMMON block workspace have to be relocated. Each of these different types will have to have a different address added into the order. Each word of code has associated with it a relocation marker. So when the code is relocated for each word of code the relocation marker for that word is looked at and the appropriate address is added into the word. The relocation markers are 3 bits for each instruction, are packed 8 to a word and appear as a block following the block of code to which they relate. Below is a table giving the interpretation of the relocation markers

0	Instruction absolute
1	Not used
2	Constants
3	Variables
4	Dynamic Routine Entries
5	Not used
6, 7	COMMON Block workspace

The first thing that Pass 2 does is to input, using block KICKOFF, the global information, which will contain such things as the number of DO loops. Then block SETITAL decides the store layout and inputs the dictionaries. After this block ALLOCA allocates the run-time workspace which are not absolute yet. In order to issue code Pass 2 must know whereabouts in the workspace a variable lies so each variable has associated with it an integer giving the location at which the variable starts relative to the beginning of the workspace.

The main body of the intermediate code is then input by block NEXT which then immediately branches when it meets a statement code. Each branch leads to a different block which processes the intermediate code and issues object code for a particular statement type. When the statement END is met the subprogram head is output to the object code.

5.2 Block ALLOCA

Block ALLOCA sets up a table known as LONATAB to hold information about simple variables and arrays. If a variable or an array has been referenced by the integer n in Pass 1 then the nth location will hold in the bottom 15 bits its position in the workspace relative to the beginning of the workspace. The top 3 bits hold the relocation marker. In the case of an array it will hold the array ZERO, an address which is explained below.

If an array has been declared in

```
DIMENSION A(D,E)
```

then if each element of the array occupies 1 location the address of A(3,4) is

address of A(1,1) + (4-1) D + (3-1)

or address of A(0,0) + 4D + 3

A(0,0) is not an element of the array A, since in Fortran subscripts start at 1, but is known as the virtual origin of the array and its address is the array ZERO. The array ZERO is used in finding the address of elements of arrays since it simplifies the code. Below is the code for above example.

```
LDR:L   3
LD:L    4
MULM    D
MULS    ELSIZE
ADD     ZERO
```

ELSIZE holds the number of locations needed to store each array element. It will be 2 for real elements, 4 for double precision and complex, and 1 or 2 for integer and logical depending on whether the array is in COMMON or not.

Thus to access an array various locations are needed to hold the array ZERO, ELSIZE and the dimensions of the array. For a formal array the dimensions will not be known until run-time and so these locations will be in workspace.

In the allocation of work space block ALLOCA scans the names in the variable dictionary. If the item is an array then the action depends on which of the 4 types of array it is. If it is a dummy array then very little is know about it and workspace is allocated. If it is a dynamic array then everything is known about the array except ZERO and a location is allocated so that this can be filled in at run time. If the array is in ARRAY STACK then the ZERO calculated relative to the beginning of the stack is put in the constants block. If the array is in COMMON then the ZERO is put in the constants block and relocation number 7 is used.

A difficulty arises when arrays are declared in COMMON which is illustrated below. If subprograms SPA and SPB contain the following COMMON statements respectively

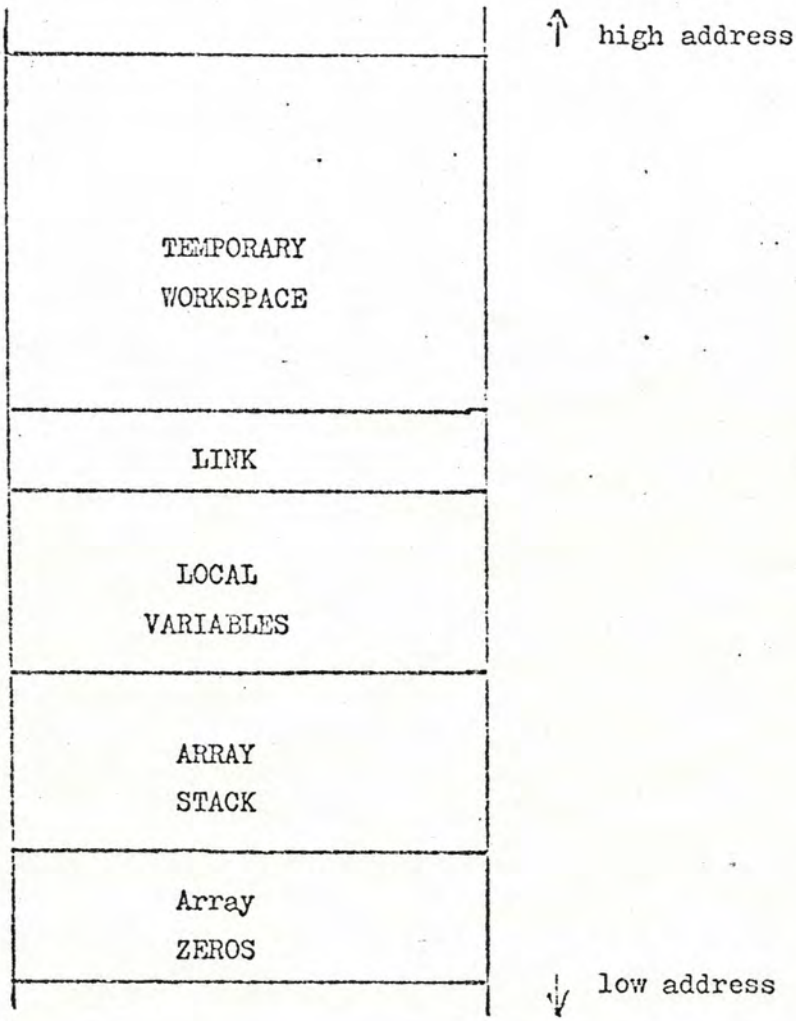
```
COMMON/BLOB/X(1000)/GLOB/Y(500)
COMMON/GLOB/A(500)/BLOB/B(1000)
```

The correspondence between the arrays is established at run time, when it will be known that the named COMMON block GLOB begins at location N, say, and the named COMMON block BLOB begins at M, say. It should not be assumed that the ZERO of the array Y lies in COMMON block BLOB. This means that in accessing the array ZERO of Y the address should be N-1 rather than M+999, which would result from the use of relocation marker 6, with associated address 999. Instead we use a word containing both the common block number and the position relative to that common block, with RLM 7.

If when scanning the dictionary the item is a local variable then space is allocated for it. If the item is a dummy variable then a location is set aside to hold the pointer which will be used at run-time. If the item is a variable in COMMON or the ARRAY stack then an entry is made in LONATAB.

If the subprogram calls any other subprograms then block ALLOCA will allocate space for the links which will be needed.

Below is a map of the workspace for a subprogram:



5.3 Label Structure

In Fortran labels are integers and they are written in the first 5 columns of a statement. If the label is less than 5 decimal digits then it does not matter where abouts within the 5 columns the label is written.

Pass 1 builds up a label dictionary as it processes the source code. Each entry in the dictionary is two words. The first word holds the label number used by the programmer. The second word is the label codeword and the significance of its bits are given below:

Bits 1-7	Not used.
Bit 8	Set if referenced by backward GOTO.
Bit 9	Set if FORMAT reference is backwards.

Bit 10	Set if referenced in GOTO or Arithmetic IF.
Bit 11	Set if referenced in READ/WRITE.
Bit 12	Set if referenced in DO loop.
Bit 13	Set if defined at FORMAT.
Bit 14	Set if label is defined.
Bits 15-24	Integer number assigned to label by Pass 1.

This integer assigned to a label by Pass 1 is such that label number N was the Nth label encountered by Pass 1. If a label is used within a DO loop then there are extra codewords in the dictionary to hold information about DO loop levels. At the end of a subprogram block CEND checks to see if all the labels have been defined.

The Pass 1 label dictionary is not passed over to Pass 2, but Pass 2 builds up its own label dictionary. Each entry in this dictionary consists of one codeword and the dictionary is a look-up one. The codeword for label number N will be the Nth location in the dictionary. The significance of the bits in codeword is given below.

Bits 1-18	Give the offset to the switch table entry if backward reference or the position of the label if already in object code.
Bit 19	Set if there are any references to label in READ/WRITE statement.
Bit 20	Set if there are any backward GOTO references to label.
Bit 21	Set if there has been a backward reference to label in READ/WRITE statement.
Bit 22	Set if there has been a backward GOTO reference to label.
Bit 23	Set if label has been met yet.
Bit 24	Set if label is defined.

The following 5 statements illustrate the 4 different types of jump to a label.

```
      READ (3,10)
      GOTO 10
10    FORMAT (.....)
      READ (3,10)
      GOTO  10.
```

When either of the first 2 statements is met by Pass 2 the object code for the statement with label 10 has already been issued and hence it is possible to issue the object code for them since the distance between them and the statement with label 10 is known. When, however, the last 2 statements are met the compiler has no idea where the object code for the statement with label 10 is. The distance between them and the FORMAT statement is not known until the FORMAT statement is met. For the last 2 statements a jump forward to the switch table is issued in the object code. The switch table is situated at the end of the object code (i.e. is output first by Pass 2). When the FORMAT statement is met the appropriate entry in the switch table is filled in with a jump back to the label.

There is an added complication with FORMAT statements since they can be jumped to in 2 different ways. The first way is by a READ/WRITE statement when we want the object code for the FORMAT statement to be obeyed. The second way is by a GOTO statement when we want the effect to be that the next statement to be obeyed is the one following the FORMAT statement. Thus for a FORMAT statement there are 2 entries in the switch table to deal with these 2 cases.

To sum up, (i) the label block in Pass 2 enters with the label number from the intermediate code and exits with the offset to the label in the object code or the switch table (ii) when a label definition is met the entry in the dictionary is filled in and if it has a switch table entry then that is also filled in.

All the labels in an assignment GOTO statement have switch table entries and a statement of the form:

ASSIGN 10 to I.

sets up I to point at the switch table entry for the label 10.

5.4 Arithmetic and Logical Expressions.

Pass 1 has converted the source code for arithmetic and logical expressions to reverse Polish form and has output this in reverse order to the intermediate code. Thus Pass 2 reads the reverse Polish forwards.

When Pass 2 gets an operand it pushes it down on to the compile time stack (CTS). When it gets an operator it pops up the top 2 items of the CTS, issues code for that operation and pushes down the result onto the CTS. This method as it stands leads to inefficient code as the following example illustrates:

Y = (A+B) * C	Source code
YAB + C * =	Reverse Polish
FL A	Object code.
FA B	
WF X1	
FL X1	
FM C	
WF X2	
FL X2	
WF Y	

To make the code more efficient the flag RTA is used in conjunction with the entries in CTS. Below is the structure of an entry in the CTS.

Bits 1-15	Address of operand.
Bit 16	Size of element in array.
Bits 17 & 18	Address modification bits (i.e. direct, indirect, modified or literal).
Bits 19-21	Type of operand.
Bits 22 & 23	Type of workspace.
Bit 24	Set if result in run-time accumulator.

The run-time accumulator is the M register, floating-point, double precision or complex accumulators depending on the type of the operands being manipulated at any given time. After an operation the result will be in the run-time accumulator and the result entry packed down on the CTS stack will have bit 24 set to signify this. The flag RTA will be set up to point at this result entry. If the value of a result or an operand is not held in the run-time accumulator then bit 24 of that entry in the CTS will be 0. The purpose of all this is to remove redundant store and load instructions in the object code since when it pops up an item from the CTS it has no need to load that item if bit 24 is 1.

The code issued for the above example now becomes:

```
FL    A
FA    B
FM    C
WF    Y
```

Since Pass 2 gets the intermediate code forwards the code is not issued directly. Instead it is pushed down on the issue stack and at the end of the statement is output in reverse order.

When a result is held in a run-time accumulator and the corresponding result entry has bit 24 set, and then that particular run-time accumulator is needed in a subsequent operation, code is issued which stores the run-time accumulator away in temporary workspace X, say. An entry for X with bit 24 zero is put in the CTS to overwrite the previous result entry. The flag RTA will then be set to zero to signify that no item in the CTS has its value in the run-time accumulator.

When Pass 2 wants to put the address of the variable NAME into the M register the following code is obeyed:

```
LDR       NAME
ADDR      LONATAB
LD:M      0
AND:L     B:1:15
```

Below is the code to put the address of the element A(I,J) in TEMP where d and e are the dimensions if known literally, and D and E if they are identifiers.

```
LD        J
LDR       I
MULM      D  (or MULM:L   d)
LDK       A SIZE (=0, 1 or 2)
JZ:S      1
SML
ADD       ZERO
ST        TEMP
```

All constants are treated in much the same way as identifiers and are put in a dictionary. This leads to some wasted space as an integer constant between 0 and 32767 is compiled as a literal instruction.

Below are some examples of the object code issued for some operations.

(i) Unary Minus (e.g. -A)

```
ST      C or B
LD      A
NADD:L  0
```

Here the CTS contains A, B and C with A at the top and C at the bottom. If RTA points to B or C then all 3 instructions are issued. If RTA is zero then only the last 2 instructions are issued. If RTA points to A then only the last instruction is issued. In the first case C is not loaded back into the relevant run-time accumulator after the operation. If the operands have different types then the necessary type conversions will have to be made.

(ii) A.LT.B

```
FL      A
FS      B
FSIG
JN:S    3
LD:L    0
JP:S    2
LD:L    1
```

This piece of code sets the M register to 0 if the logical relation is false and sets bit 1 if it is true, this being the interval representation of the 2 logical values.

6. Functions.

Functions, either function subprograms or statement functions, are used in Fortran to avoid having to write pieces of code which are basically the same, many times in a program. A function subprogram has the following structure:

```
FUNCTION    X(A)
X = A
RETURN
END
```

A function subprogram must have a value associated with it. This means that the name of the function must appear on the left hand side of an assignment statement. A statement function, however, appears in the subprogram in which it is needed and consists solely of the assignment statement which assigns the value to the name of the function.

When the value of a function is required the name of the function together with the parameters are written in the arithmetic expression. Unlike a subroutine, a function must have at least one parameter. In the definition of the function the parameters are formal parameters while in the execution of a function the formal parameters are replaced by the actual parameters for this particular execution of the function. This replacement of formal by actual parameters is achieved using indirect addressing. For each formal parameter there is reserved one location in the constants block to hold the address of the actual parameter. When the value of the function is required the addresses of the parameters are put in the constants block. During the execution of a function when the value of an actual parameter is needed the address held in the constants block gives the location of the parameter.

If an actual parameter is an array element the address held in the constants block may have to be modified from that of the array element to that of the array ZERO if it is found that during the execution of the function the whole array is needed. The calculation of the array ZERO from the address of an element may be complicated by the array having variable dimensions and by its element size depending on the type of workspace the array is in. A real element occupies 2 locations, a complex or double precision element occupies 4 locations, while an integer or logical element occupies either 2 or 1 location depending on whether it is in COMMON or not.

7. Standard Function Generation.

The routines for evaluating standard functions are part of the Dynamic routines which are only used at run time.

The routines were basically designed for the 4120, and consequently they involve a lot of integer working and they make use of the fact that on a 4120 multiplication and division take about the same time. However on a 4130 multiplication takes about half the time for a division.

We must first of all make a distinction between function approximation and function generation. Function approximation involves fitting an approximating curve to the function over a relatively small range while function generation involves evaluating the function approximately over a relatively large range.

The first step in generating a function is to reduce the range of the function using interval reduction techniques. Thus we are looking for a function f(x) where

$$x \in \left[\frac{-1}{g(k)} , \frac{1}{g(k)} \right]$$

where g(k) is either 2^k or K. The larger we make K the smaller the interval about the origin becomes. As an example I shall use the function LOG with a real argument.

i.e. LOG $(a \times 2^b)$

The argument here lies between 1 and the largest floating point number. If the argument was orginally between 0 and 1 then it is inverted and the function negated at the end.

$$\begin{aligned} & \text{LOG } (a \times 2^b) \\ = \quad & (b-1) \text{ LOG } (2) + \text{LOG } (2a) \\ = \quad & (b-1) \text{ LOG } (2) + \text{LOG } \left(\frac{2^k + n}{2^k}\right) + \text{LOG } \left(\frac{(2^{k+1}) \times a}{2^k + N}\right) \end{aligned}$$

Since 2a lies between 1 and 2 we have divided this range into units of $\left(\frac{1}{2}\right)^k$ where:

$$1 + \frac{n}{2^k} < 2a < 1 + \frac{n+1}{2^k}$$

The above expression for LOG $(a \times 2^b)$ is exact and the last term is the only place at which errors can occur at run time since n and k will be determined by the accuracy to which the function is required. The range of the argument of the last term is 0 to $\left(\frac{1}{2}\right)^k$.

To approximate a function in a small region we could have used a truncated Taylor series but we did in fact use a truncated rational approximation with n terms in both the numerator and the denominator and with the constant term in the denominator equal to 1. The expansion of this is the same as a Taylor series with 2n terms. Errors will be introduced due to the neglected terms.

The rational approximation function is then converted into a continued fraction of the form:-

$$A + \cfrac{x}{B + x + \cfrac{C}{D + x + \cdots}}$$

On a 4120 a continued fraction with 2n terms is quicker than a Taylor series with 2 terms since the Taylor series requires 2n additions and 2n multiplications while the continued fraction requires 2n additions and n divisions.

Round off errors cannot be ignored in this sort of work. For an oscillating Taylor series round off errors can be very bad. The propogation of round off errors up the continued fraction are not serious and can be minimised by having the interval sufficiently small that the continued fraction is nearly constant.

If the function is a contraction mapping (i.e. an interval gets mapped onto a smaller interval) then no extra inaccuracies will be introduced because the function is a contraction. If, however, the function is an expansion mapping (i.e. an interval gets mapped onto a larger interval) then extra inaccuracies will occur since many values of the function will not be obtainable for any value of the argument. If

$$f(x + \varepsilon) = f(x) + \delta .$$

we can put ε equal to 2^{-39}, this being the inaccuracy which results from representing a floating point number in binary form. For any function the constant A in the following equation can be evaluated

$$A\frac{(\varepsilon)}{(x)} = \frac{\delta}{f(x)}$$

For a contraction mapping $A < 1$ and so the inaccuracy of the function will be less than 2^{-39} (i.e. no loss of accuracy). For an expansion mapping there will be a loss of accuracy. If $A=2^p$ then we will lose p bits from the bottom of the

mantissa of the floating point number. To illustrate this point I shall use the example above of the LOG function

$$\text{LOG}\ (x+\varepsilon) - \text{LOG}\ (x) = \delta$$

$$\text{LOG}\ \left(1 + \frac{\varepsilon}{x}\right) = \delta$$

$$\left|\frac{\varepsilon}{x}\right| = |\delta|$$

$$A = \frac{1}{\text{LOG}\ (x)}$$

Thus when $x > e$ the mapping is contraction. Otherwise it is an expansion mapping and as x tends to 1 the inaccuracy increases.

To sum up, when a routine is needed to generate a particular function, an interval reduction technique is worked out. Then the continued fraction for that function is worked out. The number of terms needed in the continued fraction will be determined by the accuracy to which the function is required. The constants in the continued fraction together with the number of terms is stored in the Dynamic routines. At run-time when a standard function has to be evaluated the relevant routine is entered with the parameters. The appropriate constants are then found and the continued fraction evaluated. The routine exits with the value of the function.

A small number of the standard functions are evaluated using iterative techniques, the most important being for square roots. To find the square root of the number a the following iteration is used

$$x_{n+1} = \tfrac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

An interval reduction is done first of all so that a belongs to $\left[\frac{1}{4},1\right)$. This means that the square root of a belongs to $\left[\frac{1}{2},1\right)$. The initial approximation is taken to be

$$x_o = a/2 + C_i$$

where a belongs to $\left[a_i, a_i+1\right]$ and a_i and C_i are constants which can be written in the form

$$a_i = \frac{i}{2^j} \quad , \quad C_i = \frac{P}{2^k}$$

where i, j, P and k are integers. If the iteration gives a result which is accurate to s bits then another approximation will give the result accuracte to 2s bits.

8. Relocatable Code and Run-time Store Use.

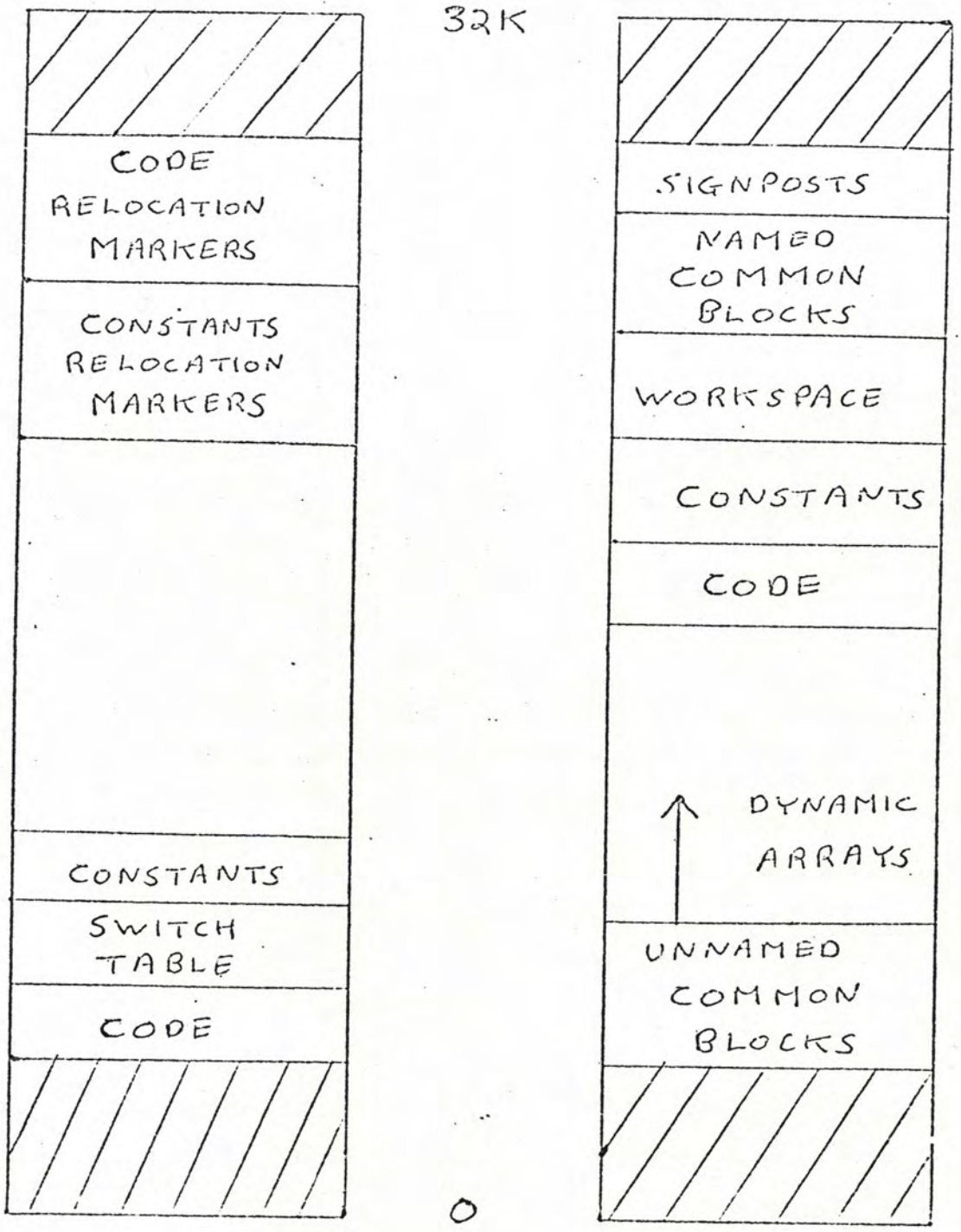
Under the control of the Executive the source code for a subprogram is input and processed by the compiler and relocatable object code is produced. This will be on magnetic tape in the non-basic system and on paper tape or in store in the basic system. For a system with a library facility this will hold on magnetic tape the relocatable object code for Neat, as well as Fortran, subprograms.

The relocatable object code consists of relocatable orders and constants, symbolic information such as the subprogram name, subprograms used and COMMON blocks used, and the amount of space needed for local workspace and dynamic arrays.

When a program has to be run the program LOADER is used to transform the collection of blocks of relocatable object code for the various subprograms of the program into a form which is ready to run. There are 2 different LOADERS, the functions of which are described below.

Basic LOADER

The basic LOADER is under 1000 locations long and will only make use of 32K of store. It will only load relocatable object code from store or paper tape. It has only rudimentary library facilities for inputting subprograms from paper tape. There are no segmentation facilities and the program has no effective diagnostic facilities. Below are 2 store maps of a one-subprogram program, the one on the left just after the relocatable object code has been put in store and the one on the right when the code has been relocated.



STORE MAPS UNDER BASIC SYSTEM

When the relocatable object code is put into store the relocation markers are put at the top of free store and the code, constants and switch table are put at the bottom of free store. Space is now allocated at the top store for the subprogram signposts and workspace and for any new named COMMON blocks used by the subprogram. Space for unnamed COMMON blocks and dynamic arrays is allocated at bottom of store. Since all the absolute addresses for this subprogram are now known the code and constants are relocated below the subprogram workspace.

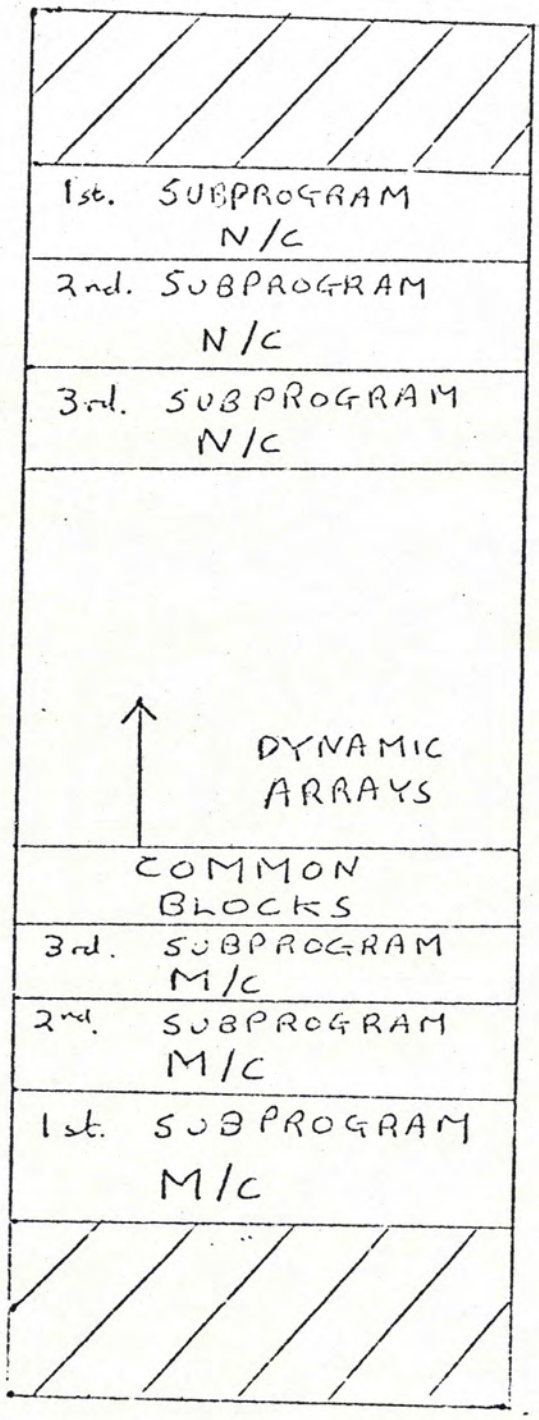
This procedure is repeated for each subprogram in the program overwriting the compiler if necessary. The LOADER checks to see if all the subprograms referenced have been brought in and also keeps a dictionary of COMMON block names.

Non-Basic LOADER

As the address part of an instruction is 15 bits long it is not possible to reference locations above 32K in an instruction. The S register is however 17 bits long which means that code can be put between 32K and 64K. It is possible to have a statement of the form

```
COMMON  X,  Y(1000)
```

where Y (1) is below 32K and Y (1000) is above 32K since the elements in the array are indexed indirectly. If however X is above 32K it will get thrown off. Below is a store map of a program ready to run.



STORE MAP OF PROGRAM READY TO RUN UNDER NON-BASIC SYSTEM.

With the non-basic LOADER all the relocatable object code for the program to be run is input to store from magnetic tape. The store map is then worked out followed by the relocation of code and constants. In the store map above we have used the notation N/C to stand for Normal Chapter which contains the code of a subprogram, and the notation M/C for Main Chapter which contains the workspace constants and signposts of a subprogram.

The non-basic LOADER is much longer than the basic one and provides extra facilities probably the most important of which is that of segmentation. If a programmer knows that his program is too big to fit into the store available he can split his program up, using the SEGMENT statement between 2 subprograms. At run-time not all the segments would have to be in store at any one time, the others being held on magnetic tape and brought into store when needed if necessary, over-writing unwanted segments.

The library facility enables subprograms to be filed away on magnetic tape and to be used by another program at a later date using the FETCH statement.

The non-basic LOADER is also used to input systems programs from magnetic tape. The systems programs have the same format with their Normal Chapter at the top of store and their Main Chapter at the bottom of store.

9. Dynamic Routines

These routines are only used at run-time and are entered by a JIL instruction via a switch table. Below are descriptions of the various types of routines.

1. Standard Mathematical Function Routines.

   The mechanism of these routines has already been described.

2. Exponentiation.

   The code for the exponentiation operator is not put into the object code but the dynamic routines are used to evaluate the exponentiation operation. A different routine is used for each combination of operand types i.e.

   R1 ** R2 = EXP(ALOG(R1)*R_2)

   I ** R = EXP(ALOG(FLOAT(I))*R)

   R ** I = R ** (100101) = $R^{32} * R^4 * R$

3. There is a routine which picks up the address of parameters and hands them over to functions and subprograms. The block PARAM is used to hand over parameters from a Fortran subprogram to a Neat subprogram.

4. At the beginning of a subprogram a routine is used to allocate space for dynamic arrays. Block STALLOC allocates the dynamic array space while block STDEL deletes the dynamic array space.

5. PAUSE routine.

   This routine, when a PAUSE statement is encountered during the running of a program, outputs the following message to the typewriter.

   ** PAUSE

When a full stop is typed the program continues running.

6. Magnetic Tape routines.

   Other than READ/WRITE statements there are the 3 statements BACKSPACE, REWIND and ENDFILE which are catered for in the Dynamic Routines.

7. Run-time error block.

   If an error occurs during the execution of a READ/WRITE statement then the following message is output to the lineprinter.

   RUN ERROR n

The current FORMAT statement and the line of data, if the error occurs on input, are also output, in the non-basic system.

When overflow or underflow occurs in a floating point number the message

FPOFLO

is output and the run continues with the appropriate default value. When this has happened 10 times the run is abandoned. The default value for underflow is zero and for overflow is either the largest possible positive real number or the smallest possible negative number.

8. READ/WRITE routine.

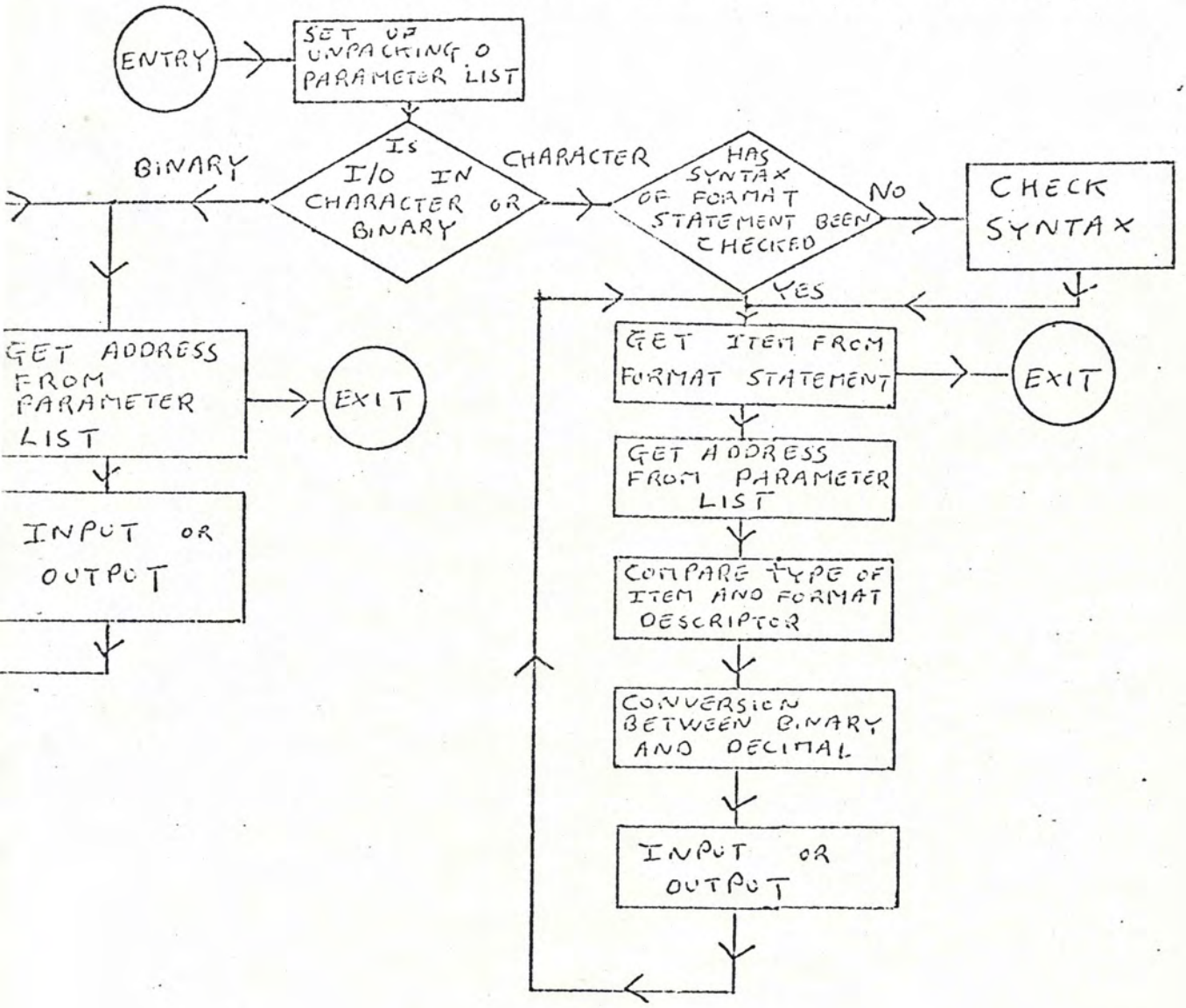
One of the major functions of the Dynamic Routines is the input and output of data. The routine is entered with the M register pointing at the READ/WRITE parameter list and the R register pointing at the packed FORMAT statement, both of which are held in the constants block. The packed FORMAT statement is essentially in the same form as in the source code. The parameter list is however very different from its form in the source code. The list begins with the device number and similar information. Following this is coded information about identifiers, arrays and DO loops in the READ/WRITE statement. The parameter list for the 2 statements.

```
DIMENSION A(10,20)
READ (3,9)  A
```

would look like:

```
DO begin I
    1
DO begin J
    1
ZERO of A
Dimensions of A
    I
    J
DO end J
   10
    1
DO end I
   20
    1
```

Below is a general flow chart of the input/output routines:



Flow chart of input/output routines.

Decimal - binary conversions

A decimal floating point number has its mantissa held in a 12 word array, with 1 digit per word. On output of a number we have to convert a binary number of the form

$$a \times 2^b \quad \text{where } \tfrac{1}{2} \leq a < 1$$

to a decimal number of the form

$$c \times 10^d \quad \text{where } 1 \leq c < 10$$

i.e. log (a) + b log (2) = log (C) + d.

As a provisional value for d we can use b log (2), where

$$\log (C) - \log (a) \in (0, 1.3010 \ldots\ldots)$$

Thus

entier (b log (2)) = D

where D is d or (d+1).

Now $c \times 10^d$ is divided by 10^{1D1} to obtain c if D=d or C/10 if D=d+1. This value of c is multiplied by 10^5 if D=d or 10^6 if D=d+1. A FENT instruction followed by 5DIVM instructions will give the 6 most significant bits of c. The remaining 6 digits are obtained by subtracting the number obtained so far from the original number, multiplying the result by a power of 10 and the performing a FENT instruction followed by DIVM instructions. For a double precision floating point number the next 12 digits are obtained in a similar way. In this paragraph the description has been for positive d. For negative d interchange the operations multiply and divide.

10. Executives.

The Executive is the systems program which organises the input, compiling, loading and the running of a program. As with the LOADER there are two versions, one for the B20 system and one for the T30C system.

The B20 system is paper tape orientated and does not use, or provide facilities for the use of, cards or magnetic tapes. All thesystems programs, compiler and programs to be run are input from paper tape. The system will only run programs written in Fortran.

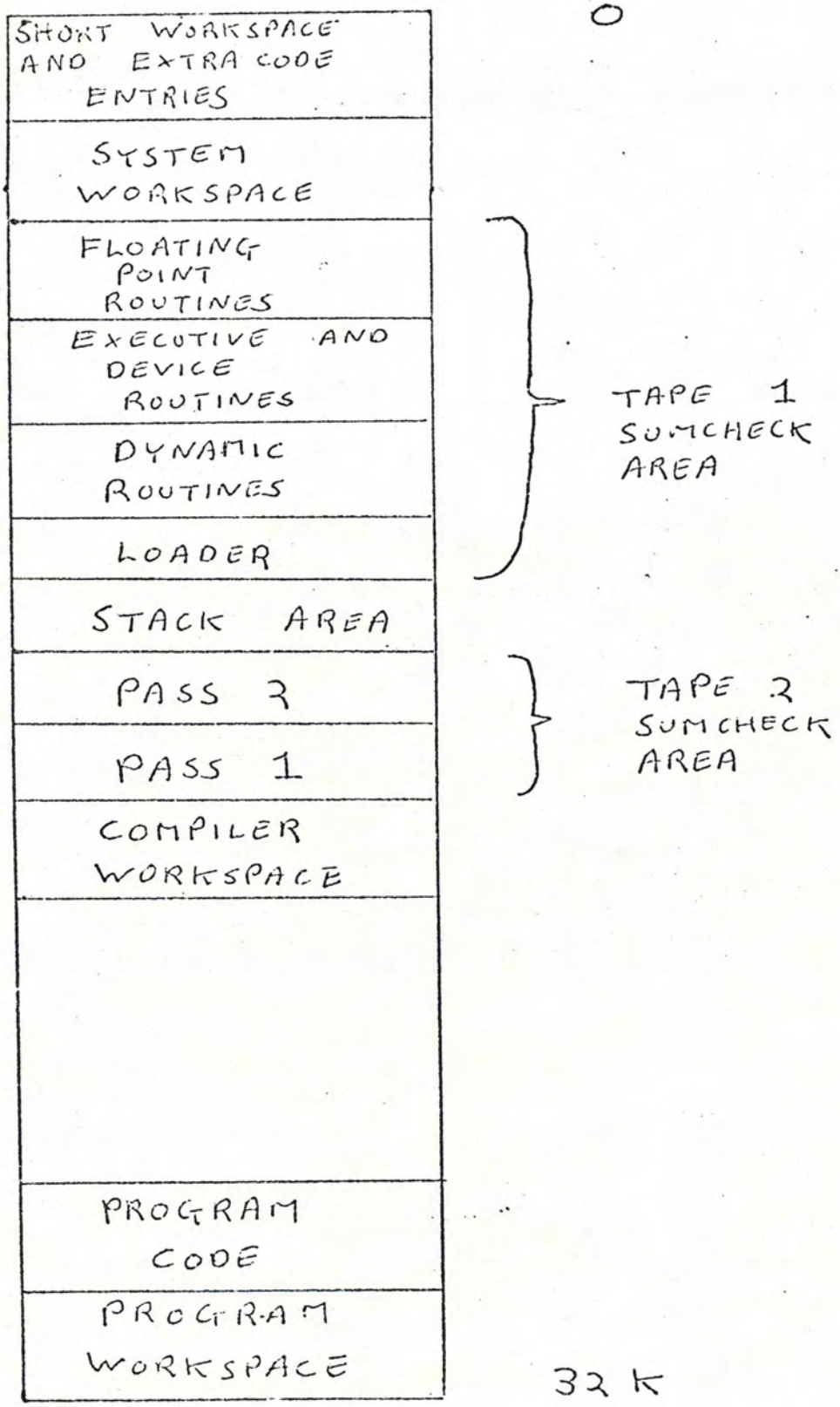
The T30C system, however, provides many more facilities than the B20 system. It is magnetic tape orientated and all the system programs and compilers are held c magnetic tape. Programs in Fortran, Algol and Neat can be input from paper tape, cards or magnetic tape. The library facilities enable subprograms to be filed on magnetic tape as relocatable object code which can then be loaded to make up a subsequent job. Whole batches of jobs can be run without any operator intervention.

The B30C system is an extension of the B20 system. There are 2 main differences between them. The first is that programs can only be input from cards although the systems programs and the Fortran compiler are input on paper tape. The second difference is that it is possible to run T30C card jobs since the B30C system ignores any T30C control card which is not a B20 control card.

Below are descriptions of the B20 and T30C Executives:

B20 Executive

Below is a map of the store after a program has been compiled under the B20 system.



STORE MAP OF B20 SYSTEM.

The system is issued as 2 types, the first containing the systems programs and the second the compiler. Between each program the store corresponding to the 2 types is sum checked in order to try and detect any corruption that might have taken place.

The store map above is for when a program has just been compiled. Before the program is run the compiler is copied to the top of free store. The dynamic arrays are allocated space building up from the bottom of free store. It is possible for the dynamic arrays to overwrite the compiler. In this case before the next program is compiled the compiler will have to be reinput. If the compiler has not been overwritten then it is copied back down to the bottom of free store before the compilation of the next program.

To try to cut down operator control in the running of jobs, as well as the actual Fortran statements themselves there are also control statements. In B20 there are 3 control statements which are shown below:

&JOB; <job code>; <job name>;
&LIST;
&RUN;

Control statements have the ampersand in the first column of a line.

When the Executive is first entered it looks for the compiler. If it is at the top of store it is copied down and the sum check made. If compiler was absent or had been corrupted the compiler must be reinput. After this has been done the Executive searches for a &JOB; statement. When this statement has been found the job field is output to the typewriter followed by the message

** COMPILE <

If the reply is L. the job is run load and go while if it is P. the object code of the program is output to paper tape.

The Executive now enters Pass 1. All the source code passes through the Executive where it is put in the buffer RESERVE. If the line of source code is a Fortran statement it is handed over to Pass 1, while if it is a control statement the Executive will act upon it. Thus the compiler does not know about the existence of control statements.

Pass 1 builds up the intermediate code in free store until an END statement is met. Control is then handed back to the Executive which now enters Pass 2. The intermediate code is read by Pass 2 and the object code put at the top of store. Each subprogram is dealt with in this way until an &RUN; statement is met. The Executive now outputs the message

** RUN <

The typing of a full stop causes the compiler to be copied up to the object code and the program run.

If the compilation of a program fails or an error occurs in the loading or running of a program the job is abandoned and the Executive searches for the next &JOB; statement. At any time control can be transferred to the Executive by pressing the message key, which causes the message

** <

to be displayed on the typewriter and the Executive waits for a 4 letter reply. If the reply is ABAN the job is abandoned and the Executive searches for the next &JOB; statement. If it is STOP the run is abandoned and the Executive searches for the next &RUN; statement, while if it is CONT the running of the job continues as though the message key had not been pressed.

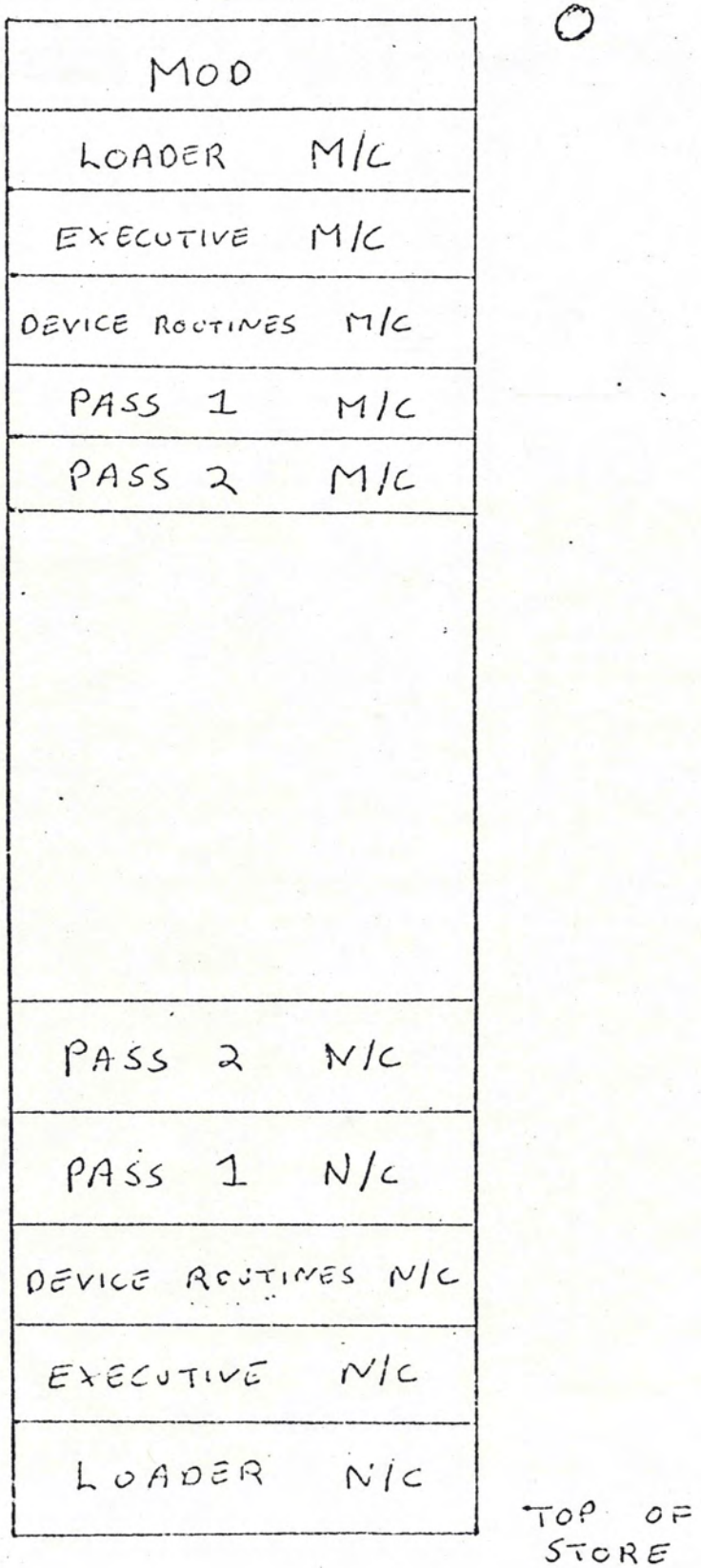
The &LIST; statement causes the subsequent source code statements of that job to be listed on the lineprinter. Under B20 a job will have the following format:

&JOB;
&LIST;
<source code>
&RUN;
<data>

T30C Executive

The T30C Operating System is magnetic tape orientated and all the systems programs and compilers are held on magnetic tape on handler 0. They are brought down from magnetic tape to store by a binary LOADER. Below is a map of the store just before the compilation of a program.

492 follows



STORE MAP OF T30C SYSTEM

As well as Fortran programs the T30C Operating System will also run Algol and Neat programs. To run Algol and Neat programs different systems programs and compilers are needed and these are also held on magnetic tape. The only program which is common to the 3 systems is MOD, which contains CLOCK, CDR, EXTYPE, BLP and MT device routines. Below is a description of the Fortran Executive in T30C for a store of at least 24K.

The T30C system has many more control statements than B20 which enable whole batches of jobs to be processed without any operator intervention. Below are listed the extra control statements:

&FORTRAN;
&ALGOL;
&NEAT;
&NORUN;
&FILE;<handler number>;<filename>;
&LOAD;<Prog. name>;<handler no.>;<filename>;(<language>;)
&OPERATOR; message to be O/P ;
&WAIT;<message to be O/P>;
&UNLIST;
&TIME;<max. time>; (<expected time>;)
&LINES;<maximum no. of lines>;

When the Executive is first entered it searches for an &JOB; statement. When this has been found the clock routine is initialised. If the next control statement is &ALGOL; or &NEAT; a systems swop is initiated. If the next one is &FORTRAN; the Executive checks to see if the compiler is in store. If it is not it is brought down from magnetic tape. The compiler is now entered and it keeps the intermediate code it produces in store but writes the object code to magnetic tape on handler 1 in 511 word blocks. Each subprogram is processed by both passes of the compiler before the next subprogram is processed. When an &RUN; statement is met an end of file marker is written after the object code. The Executive now enters the LOADER which loads the program into the free store with the Normal Chapter at the bottom and the Main Chapter at the top. The program is now run. A job is terminated by the &END; statement and the Executive then searches for the &JOB; statement of the next job.

With a small store of under 24K there are some differences in the way the Executive organises the running of a program which are given below. The store map just before the beginning of the compilation of a program is essentially the same as the one given at the beginning of this section except that Pass 2 and the DRS are not in store. Each subprogram is first processed by Pass 1 and at the end of each subprogram the intermediate code is copied from store to magnetic tape on handler 2. Now Pass 2 is brought down from magnetic tape to overwrite Pass 1 and the object code is produced as before on handler 1. Before the program can be run the DRS are brought into store, overwriting Pass 2. When the &FORTRAN; statement is met Pass 1 is brought in overwriting the DRS.

Thus under T30C a typical job has the following format:

```
&JOB;
&FORTRAN;
&LIST;
<source code>
&RUN;
<data>
&END;
```

The &LIST; statement causes the source code between it and either an &UNLIST; statement or the end of the source code to be listed on the lineprinter.

This continual exchanging of Passes 1 and 2 and the DRS is necessary since in a small store there would not be room to run a program if they were all in store at the same time.

If a program is to be filed away then the following job will do this:

```
&JOB;
&FORTRAN; <program name>;
&FILE;<handler no.>; <file name>;
<source code>
&NORUN;
&END;
```

The &NORUN; statement causes the object code to be copied from handler 1 to the specified handler with specified file name. To run this program at a later time the following job is used.

```
&JOB;
&LOAD; <prog. name>; <handler no.>; <file name>;(<language>;)
&RUN;
<data>
&END;
```

In the absence of the language field in the &LOAD; statement it is assumed that the language is the same as that of the previous job.

Each job is timed and the time taken by that job so far is output to the lineprinter whenever a control statement is met. Each installation has a default time and a maximum time, which is longer than the default time. A job will be abandoned when the default time is exceeded. The default time may be changed by the programmer using &TIME; statement. The new default time becomes the programmer's maximum time plus the time taken so far. If the programmer's expected time (if present, in the statement) plus the time taken so far exceeds the installation maximum time the job is abandoned immediately. A job will be abandoned when the installation maximum time is exceeded.

Each installation has a maximum number of lines that can be output to the lineprinter in any one job. If this number is exceeded in a particular job then the job is abandoned. The maximum number of lines can be reset by the programmer by means of the &LINES; statement.

It is possible for the computer operator to change the installation default and maximum time and the installation number of lines by means of the &SET; statement. This is the only control statement which is not listed on the lineprinter since the statement is of no concern to the programmer.

11. Device Routines

When a systems program requires the input or output of information they will make a call on the Device Routines to actually deal with the peripheral device. The call will be of the form:

```
LDR     <device number>
JIL     £READ
```

and it will input either 1 character in internal code or 1 word of binary into the M register depending on the value of the variable PEAUDOUCE. In what follows WRITE can be substituted for READ where appropriate. Whether the output is binary or character depends on the value of the variable STROKE.

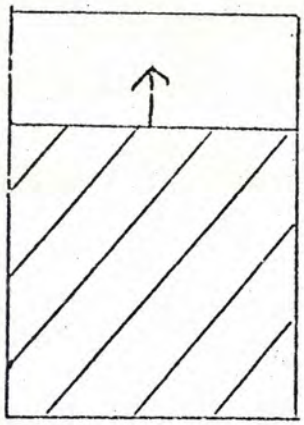
In the B20 system there is a program called DEVICES which contains the device routines for all the peripheral devices used by the B20 system. These peripherals are the 2 paper tape readers, the 2 paper tape punches, the line-printer and the typewriter.

In the T30C system the program MOD (the only program common to both the Fortran and the Algol systems) contains the routines for the card reader, magnetic tapes, lineprinter and the real time clock, and handles interrupts and attentions. The Executive, however, deals with the paper tape station and also handles some of the magnetic tape buffers.

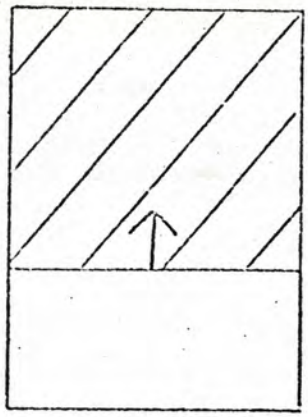
Below is a description of the device routines in the B20 system. The 4 ways the routines can be entered are by a JIL £READ or JIL £SETREAD instruction or by a device generating an interrupt or an attention. Each peripheral device has its own routine to deal with it.

Paper Tape Readers

The paper tape reader 1 is the most important input device in the B20 system since it is used for all compilations. Thus the main concern of its routine is that the transfer rate is high. To achieve this interrupts have been used in conjunction with 2 buffers as illustrated in the diagram below.



Interrupt buffer
being filled
by interrupts



Program buffer
being emptied
at program level

When the reader is ready to make a transfer it generates an interrupt and the routine puts the item in the interrupt buffer. If the interrupt buffer is then full, interrupts are forbidden. If, however, the item was a haltcode then HALT FLAG is set, interrupts are forbidden and the interrupt buffer is made to look as though it is full. By a haltcode is meant both the character haltcode and the binary representation of the integer 20, which look exactly the same on paper tape.

When a program wants an item from the reader the next item is taken from the program buffer. Before the item is taken the buffer is checked to see if it is empty. If it is then the routine waits until the interrupt buffer is full. The 2 buffers are then swopped and the next item is taken from the new program buffer. Since the new interrupt level is empty interrupts can now be allowed again. After every item has been put in the M register HALTFLAG is checked. If it is not set then there is no haltcode in the buffers and the routine exits. When HALTFLAG is set then if the item is a haltcode the necessary work associated with a haltcode is done before the routine exits.

8 bit characters are packed into the buffers 2 to a word by the instructions:

```
ST    X
GET   X
GET   X
```

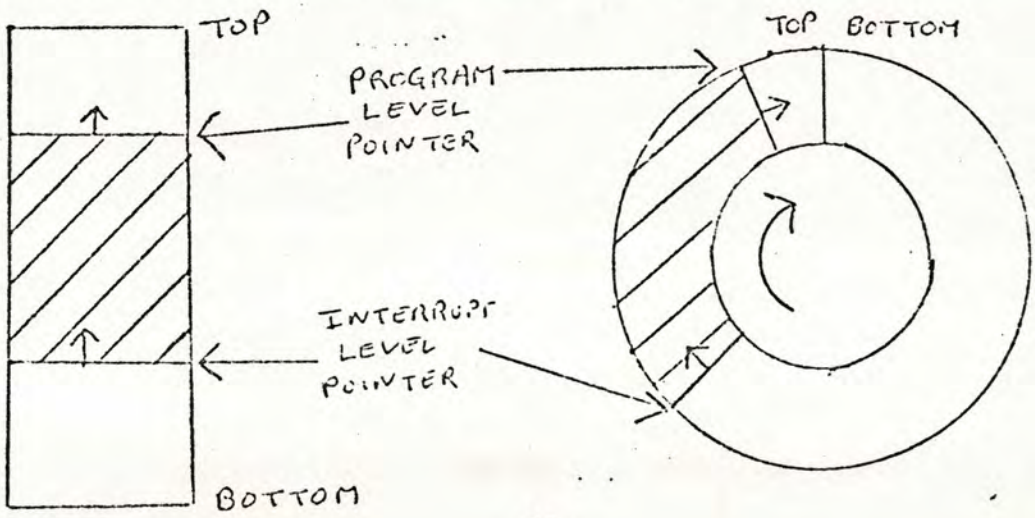
which puts the bottom half of the M register into the top half of X. It would of course have been possible to pack them 3 to a word but this would have been slower and more complicated since it would have involved shift instructions and the preserving of the K and R registers.

Since paper tape reader 2 is hardly ever used the code for its routine is short. The code simply inputs 1 character in internal code or 1 word of binary into the M register. No interrupts are used.

The Fortran compiler will typically want to input a line of source code and process this line before it looks at the next line. If a routine like the routine for reader 2 were used then the compiler would be kept waiting while the reader input the whole line. The reader 1 routine is faster since when the compiler wants a line it will be able to take a large part of it straight from the routine's buffer, which will not take so long as inputting from the device itself, leaving the reader to fill up the buffer at its own rate under interrupts while the compiler is processing a line. At the end of this section are some flow diagrams for the paper tape reader routines.

Paper Tape Punches

The 2 punches are treated by the device routines in exactly the same way. Each punch has a single buffer which is used cyclically. Characters are packed 3 to a word since the time spent packing and unpacking them becomes less significant since the punches are 10 times slower than the readers. The buffer is emptied by interrupts from the punch, the interrupt level pointer pointing at the next word to be output. The buffer is filled up at program level by the JIL £WRITE instruction, the program level pointer pointing at the next location for a word to be put into. When a pointer reaches the top of the buffer it must reset to point at the bottom. Below are 2 diagrams of the buffer.



Everytime anything is put into the buffer or removed from it a check must be made that the 2 pointers have not caught up with each other. While the 2 pointers are being compared interrupts are forbidden since if they were not an interrupt could occur during a comparison so that the comparison routine would exit with information about the pointers as they were before the comparison and not as they are at the time of exit. When the interrupt level pointer catches up with program level pointer interrupts are forbidden. When the program level pointer catches up with the interrupt level pointer the program, before it can put another item in the buffer, will have to wait for the punch to output an item from the buffer.

Lineprinter

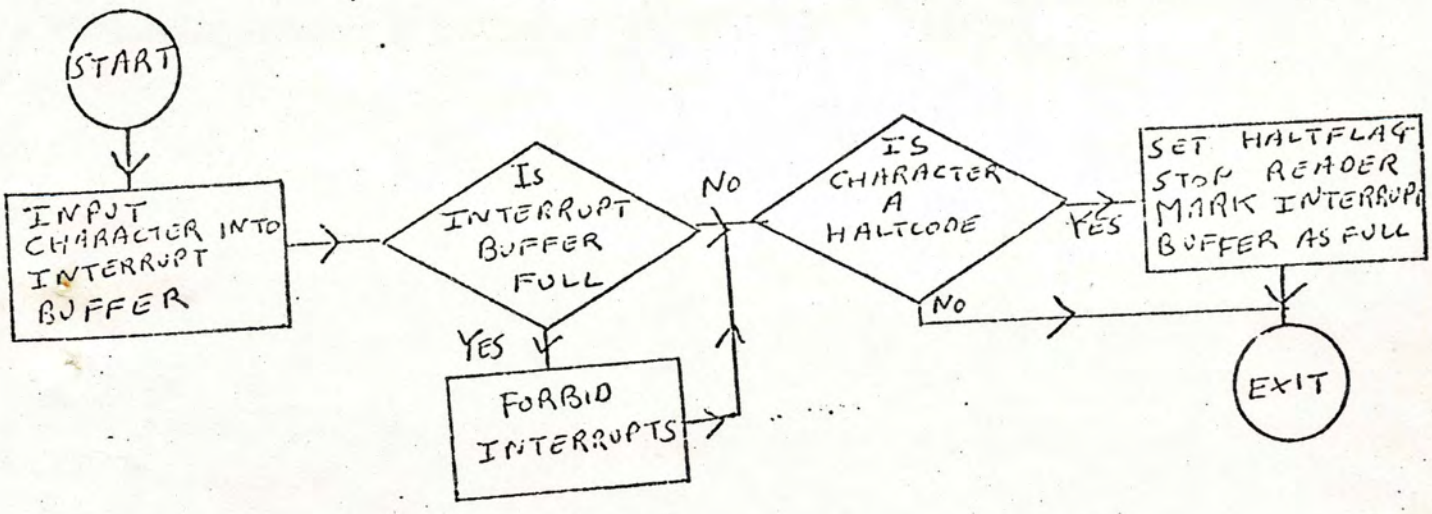
At program level the characters to be output to the lineprinter are packed into a buffer. The first character of a line is a paper throw command as shown in the table below:

space	Throw new line
0	Throw 2 new lines
1	Throw a new page
+	Overprint

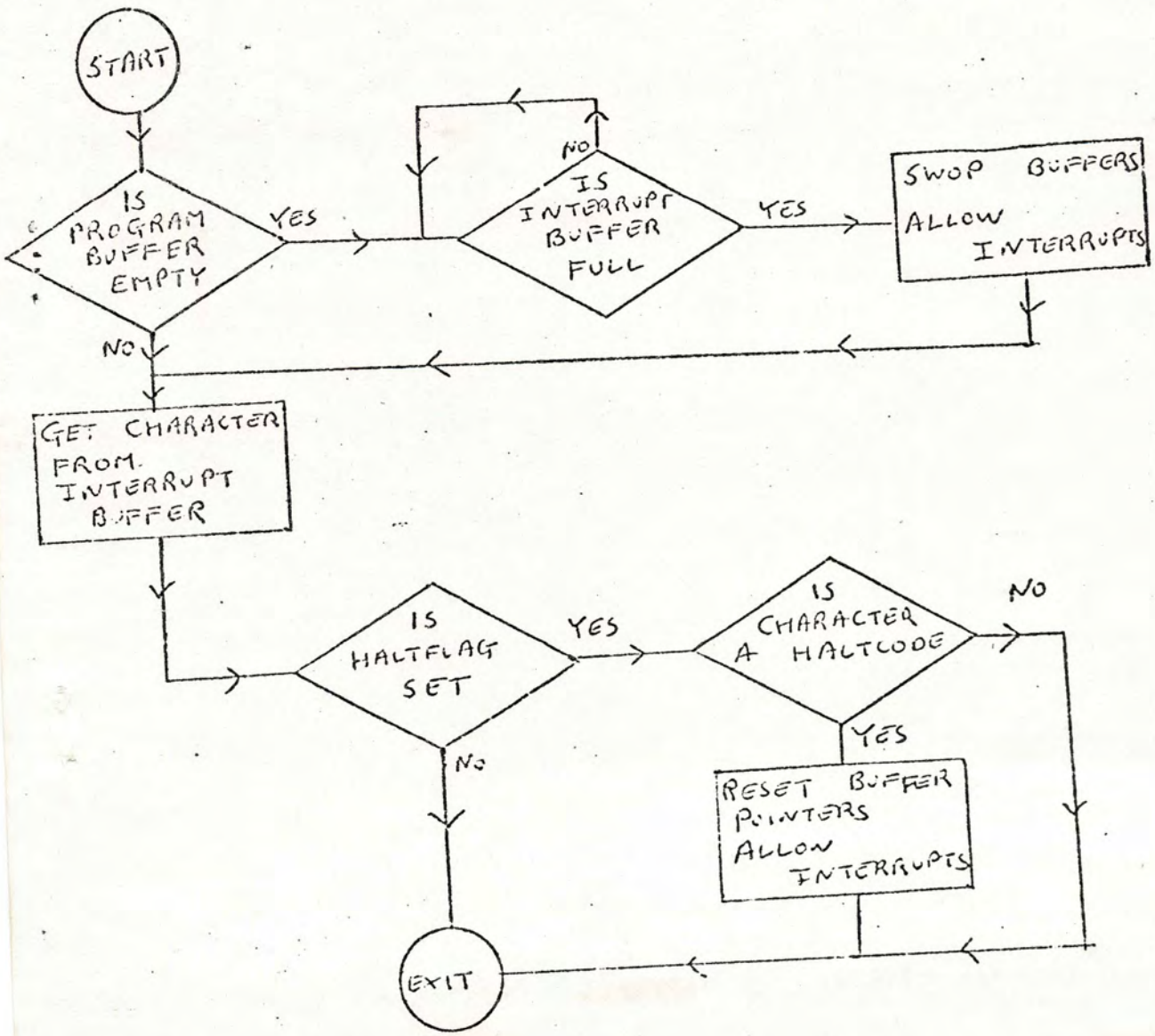
If the first character is 1 of the 4 above it is not printed otherwise a new line is thrown and the character is printed. The characters are packed into the buffer until a new line character is received or until the line is full. Interrup are now permitted from the lineprinter. When the buffer has been passed over to t lineprinter interrupts are forbidden again. If a program wants to output 2 lines in quick succession to the lineprinter it will probably have to wait awhile after the first line has been handed over for it to be actually printed before the lineprints interrupts for the second line.

Typewriter

At program level the characters to be output to the typewriter are packed into a buffer. When the buffer is full or a new line character is met the buffer is handed over to EXTYPE which then outputs the line to the typewriter.



INTERRUPT LEVEL



PROGRAM LEVEL

FLOW DIAGRAMS OF THE PAPER TAPE READER 1 ROUTINES